Mise à jour incrémentale de permissions RDF

Ethan PEGEOT, Université Claude Bernard Lyon 1, sous la direction d'Emmanuel COQUERY

2024/2025

Introduction

Motivations

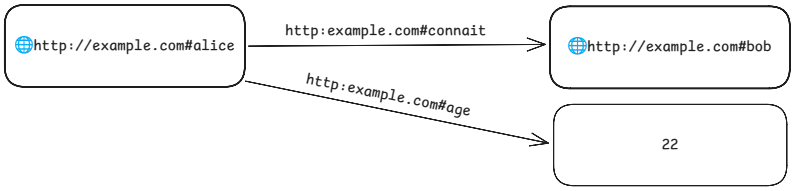
Dans le cadre de sa thèse, Tarek SAYAH a défini un système de gestion de permissions sur les bases de données RDF. L'algorithme proposé par T. SAYAH s'exécute sur l'ensemble de la base de données et ne permets pas d'insérer un triplet sans recalculer l'ensemble des permissions de la base de données.

Néanmoins, de nombreux cas d'utilisation demandent de pouvoir effectuer des opérations d'insertion et de suppression de données après leur insertion initiale.
Nous explorerons dans ce rapport l'implémentation actuelle, avant de proposer une nouvelle implémentation incrémentale et ses benchmarks

Un rappel de RDF

RDF est un système de base de données orienté graphe, dont les arrêtes sont orientées et labellées. La thématique des bases de données RDF est pilotée par le W3C et constitue un des principaux développements dans le domaine de l'Open Knowledge et dans les représentations des connaissances orientées pour la consommation par une machine.

D'un point de vue de leur représentation, les graphes RDF sont représentés par des triplets de la forme **(sujet, prédicat, objet)**. Dans un graphe RDF, la base pour représenter des données est l'IRI, forme d'URL qui a pour but d'identifier un concept. Ainsi, les sujets et les prédicats sont forcément des IRIs, là ou un objet peut être soit une IRI soit un littéral (une string, un nombre).



Un exemple de triplets RDF, visualisé
Dans l'exemple ci-dessus, `http://example.com#alice` représente le sujet, `http://example.com#connait` et `http://example.com#age` sont deux prédicats

différents, `http://example.com#bob` est un objet sous la forme d'une IRI, `22` un objet sous la forme d'un littéral.

On note qu'il existe une notation de "préfixes" qui permets des raccourcis dans l'écriture, `rdf:` étant remplacé par la partie d'URI fournie à la suite

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

Plusieurs systèmes existent pour stocker des graphes RDF. La manière la plus simple est de stocker dans un SGBDR une table contenant tous les triplets avec une colonne pour le sujet, une pour le prédicat, et une pour l'objet. Il existe également des *triplestores* natifs, qui optimisent ainsi la performance en étant spécialement adaptés à la tâche. Un exemple est **Jena TDB**, *triplestore* adossé à la librairie *Jena*, une librairie développée en Java pour manipuler des graphes RDF, référence dans le domaine.

Les triplestore peuvent être interrogés à travers un langage spécialisé, appelé le SPARQL. La syntaxe du SPARQL est librement inspirée de celle du SQL, et exprime ses conditions à travers des triplets RDF, dont les membres peuvent être des variables ou des littéraux. Une variable est préfixée par un `?`.

```
SELECT ?x
WHERE { ?x  <http://www.w3.org/2001/vcard-rdf/3.0#FN>  "John Smith" }
```

Cette requête retourne les IRIs (car des sujets d'un triple) dont le FN est *John Smith*.
L'IRI `http://www.w3.org/2001/vcard-rdf/3.0#FN` est une bonne introduction à la notion de vocabulaire. L'idée d'un vocabulaire est d'assigner une sémantique (un sens) à une IRI/un groupe d'IRI. Les IRI des vCards ont pour rapport les cartes de visites, et la propriété *FN* est un exemple de sémantique supplémentaire, donnant le sens "Full Name" à cette propriété.
Dans la suite de ce rapport, on appellera la partie suivant le SELECT la *cible*, et la partie dans les accolades les *conditions* du SELECT.

Il convient de noter qu'il existe, en plus du mot clef `SELECT`, le mot clef `ASK` qui cherche si une solution à la requête existe, et pas la solution de cette requête.

La plupart des triplestores sont en fait des quadstores, permettant de stocker une information supplémentaire sur les triplets : le nom du graphe dans lequel ils sont stockés.

La thèse de Tarek SAYAH

Dans sa thèse, Tarek SAYAH définit un système fin de gestion des permissions sur les graphes RDF. Ce système se base sur des règles définies sous une forme équivalente à une requête SPARQL `SELECT` simplifiée.

Ces règles ont la forme suivante

```
DENY ?pA <e.com#knows> ?pB WHERE
    ?pA <e.com#worksFor> ?e
    ?e <http://www.w3.org/1999/02/22-rdf-syntax-ns#class>
<e.com#GovernementEntity>
```

Cette règle par exemple interdisant de montrer les liens entre pA et pB de type `knows` si pA travaille pour e, e étant une entité gouvernementale. Le système s'appuie sur une liste de ces règles, qui peuvent rentrer en conflit entre elles et être résolues selon une stratégie choisie : appliquer la première règle dans le fichier de règles, appliquer la règle la plus restrictive, etc... (il convient de noter que les stratégies de résolution de conflits sont importantes mais en dehors du périmètre de ce projet).

Afin de stocker quelles règles s'appliquent à un triplet et lesquelles ne s'appliquent pas, un bitset est employé : l'ordre dans le bitset est le même que dans la liste de règles, le bit est allumé si la règle s'applique, et est éteint sinon. Ce bitset est sérialisé, puis stocké dans le nom du graphe du triplet.

> **Unicité d'un triplet parmi tous les graphes nommés**
>
> Même si le nom du graphe sert à stocker le bitset d'un triple, il n'est pas possible d'avoir de doublons d'un triplet dans différents graphes nommés.
> En effet, à triplet égal, règles appliquées égales, donc même quad de destination.

On appelle *Policy* un ensemble de règles pour lesquelles une stratégie de résolution de conflits est fournie. De la même manière que pour un *select*, on nomme cible la partie précédant le WHERE, et les *conditions* ou *conjonctions* la partie qui le suit.

La cible et les conditions présentent la même forme : une succession de trois éléments qui sont soit des termes, soit des variables.
Un terme est soit un littéral (un nombre, une chaîne de caractères), soit un identifiant, sous la forme d'une IRI.

Problématique

Dans le cadre de l'insertion d'un triple, deux choses rentrent en compte :

- Le calcul de ses permissions, par rapport aux autres triplets qui existent
- Le calcul des changements aux permissions des autres triplets car ce triplet ferait valider le *where* d'une règle

C'est à ce titre qu'il n'est pas trivial d'insérer un nouveau triple, et que l'approche utilisée par SAYAH est de reprendre l'ensemble des calculs à chaque insertion de triplets.

Proposition technique

Technologies employées

Il convient de noter que même si JenaTDB, le triplestore exploité à l'originer par la thèse de SAYAH et repris dans ce projet, est développé en Java et possède une API Java pleinement exploitable, il n'est pas nécessaire de développer nos briques logicielles en Java. Le choix a été fait d'utiliser Python pour ce projet, afin de profiter de sa facilité d'utilisation. JenaTDB est ainsi contacté à travers son API HTTP proposée par Fuseki.

Pour développer un parser de *policies*, nous avons exploité ANTLR, qui permets de la génération automatique de code Python exploitable.

Tout d'abord, il a fallu réimplémenter plusieurs parties du travail de SAYAH, originellement développé en Java 7. Parmi ces parties, l'implémentation d'un parser permettant de transformer les fichier de *policies* en données exploitables depuis nos algorithmes.

La grammaire de ce parser, exprimée dans la syntaxe d'ANTLR est proposée en annexe de ce rapport.

Algorithme initial

Un algorithme similaire à celui proposé par SAYAH a été implémenté dans cette version afin d'avoir une base de travail pour un algorithme dit en "one-shot", qui prends l'ensemble des triplets présents en base de données pour fournir les permissions. Le fonctionnement de cet algorithme est simple, on convertit chaque règle en une requête SPARQL, et on applique la règle directement aux triplets qui sont retournés par cette requête. Cet algorithme est simple dans son fonctionnement et peut s'exécuter sans grand coût mémoire au prix de forts temps d'exécution :

- on garde un itérateur sur la liste des triplets retournés pour chaque règle
- Pour chaque quad (le triplet + son bitset de permissions), on ajoute la règle à son bitset et on envoie une requête de mise à jour au triplestore RDF
  Ainsi, le coût mémoire de cette méthode est linéaire selon le nombre de règles (on garde un seul triplet par règle en mémoire).

Cet algorithme peut être implémenté de cette manière (avec une optimisation liée au fait de ne faire qu'une seule requête par triplet et pas une par triplet et par règle qui s'y applique)

```
Préconditions :
 - On a  un opérateur de tri bien formé entre des triplets
 - On a head : dictionnaire[règle, list[triplet]] -> list[triplet] qui
retourne la liste de toutes les têtes de liste
 - On a des opérations de requêtage sur la base de données RDF
```

```
triplets: liste de triplets à insérer
triplets_par_regle : dictionnaire[règle, list[triple]]

Insérer triplets avec un bitset nul

Pour chaque règle r
    triplets_par_regle[r] = Trier(requêter le graphe avec ce qui matche r)

Tant que head n'est pas vide: // il reste des triplets associés à une
règle
    triple_a_inserer = min(head(triplets_par_regle)) // Vu que toutes les
listes sont triées, on a un itérateur en commun

    //Avec le fait qu'on enlève les occurences de triple_a_inserer après
les avoir traitées, on peut considérer que head est le bitset de
min(head), la règle s'applique si, pour un X dans head, X =
min(head(triplets_par_regle))

    bitset : [1 si X = min(head(triplets_par_regle)) sinon 0 pour X dans
head(triplets_par_regle)]

    Modifier en base triple_a_inserer, le mettre dans le graphe lié à son
bitset

    Enlever de triplets_par_regle toutes les occurences de
triple_a_inserer
```

La conversion d'une règle en une requête SPARQL se base sur le fait que les règles sont des simplifications des requêtes SPARQL, et que pour une règle

```
GRANT ?s ?p ?o WHERE
    ?x ?y ?z
    ...
```

On peut la réécrire trivialement en

```
SELECT ?s ?p ?o WHERE {
    ?x ?y ?z.
    ...
}
```

Proposition d'amélioration

Idée générale, définitions

L'idée générale derrière l'algorithme itératif est d'effectuer le moins de calculs et requêtes à la base de données possibles lors d'une mise à jour. Il se base sur l'idée qu'il existe déjà un graphe G dont les permissions sont bien calculées, pour y ajouter une liste de triplets.

L'algorithme proposé est itératif à l'échelle d'un triple, comprendre que la fonction est de signature $G \to T \to G$, avec T le type d'un triplet et G le type d'un graphe de triplets dont les permissions sont bien calculées, une fonction qui prends un graphe dont les permissions sont bien calculées, un triplet à insérer, et retourne un graphe dont les permissions sont bien calculées avec T inséré dedans avec les bonnes permissions et les bonnes conséquences pour les autres triplets.

Pour ce faire, on distingue deux parties

- calculer les permissions de notre triple
- trouver les triplets pour lesquels une règle pourrait s'appliquer du fait de l'insertion d'un triple.

Variables instanciées ou non, le cœur de l'algorithme

Permissions du triplet à insérer

Pour calculer les permissions du triplet à insérer, nous allons trouver les règles dont la cible contient le triplet à insérer.

Dans cette version, on définit "**matcher**" comme :

```
Variables d'entrées:
 - c : tuple de 3 termes (littéraux / variables) // le triplet issu de la
règle
 - t : tuple de 3 littéraux // le triplet à insérer

Procédure Match
    Pour i entre 1 et 3
        c' <- c[i]
        t' <- t[i]
        Si c' est un littéral et que t' != c':
            Retourner Faux
    Retourner Vrai
```

Ainsi, si une règle peut matcher notre triplet à insérer, alors il faudra la prendre en compte et faire une requête pour vérifier si notre triplet est bien retourné par la règle transformée en SELECT SPARQL.

Dans ces requêtes, il est important de remplacer les variables : par exemple pour insérer le triplet

```
<http://e.com#a> <http://e.com#knows> <http://e.com#b>
```

qui **match** dans la cible de la règle

```
DENY ?x <http://e.com#knows> ?y
    ?y <http://e.com#something> ?c
```

il faudra réécrire la règle en

```
DENY <http://e.com#a> <http://e.com#knows> <http://e.com#b>
    <http://e.com#b> <http://e.com#something> ?c
```

pour prendre en compte les variables qui sont "remplies" par le triplet à insérer. Cette réécriture est faite à travers des *bindings* SPARQL.

Dans le traitement de ces situations, on appellera variables instanciées des variables dont la valeur est fixée par le triplet à insérer, et variables non instanciées les autres.

Conséquences sur les permissions d'autres triplets

Lorsqu'on ajoute un triplet, il est possible que rajouter celui-ci permette de valider certaines conditions dans le *where* d'une règle.
Ainsi, la méthode est simple : Pour toute règle r dans la policy, si r **match** le triplet dans une de ses conditions, alors on exécute la requête en liant les variables, et pour chaque triplet retourné, la règle doit s'appliquer.

Exemple

Sur la base de données suivante (en notation Turtle, la plus proche du sujet-prédicat-objet) :

```
PREFIX e: <http://e.com#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

e:alice e:knows e:bob
e:alice e:knows e:charles
e:bob e:knows e:charles
e:labo rdf:class e:governementEntity
```

Avec les règles suivantes:

```
# R1
ALLOW ?pA <http://e.com#worksFor> ?wE WHERE
    ?wE <http://www.w3.org/1999/02/22-rdf-syntax-ns#class>
<http://e.com#governementEntity>.

# R2
DENY ?pA <http://e.com#knows> ?pB WHERE
    ?pA <http://e.com#worksFor> ?wE.
    ?wE <http://www.w3.org/1999/02/22-rdf-syntax-ns#class>
<http://e.com#governementEntity>.
```

En insérant le triplet suivant :

```
e:alice e:worksFor e:labo
```

Application pour R1

Permissions du triplet à insérer

R1 matche la *cible* de R1, on **bind** donc `pA` à `e:alice`, `wE` à `e:labo`, on exécute la requête sous forme d'un `ASK` et cette requête nous retourne qu'il existe un résultat. Ainsi, R1 s'applique à notre triplet.

Conséquences sur les permissions d'autres triplets

Aucune *condition* de R1 ne matche le triplet, ainsi, cet ajout de triplet n'a pas de conséquence sur d'autres triplets déjà en base du point de vue de R1.

Application pour R2

Permissions du triplet à insérer

Le triplet ne matche pas la *cible* de R2, la règle ne s'applique pas à ce triplet.

Conséquences sur les permissions d'autres triplets

Le triplet matche `?pA <http://e.com#worksFor> ?wE.`, ainsi, on **bind** `pA` à `e:alice`, `wE` à `e:labo`, et on exécute la requête. Cette requête retourne

```
e:alice e:knows e:bob
e:alice e:knows e:charles
```

Ainsi, R2 s'applique à l'ensemble de ces triplets.

Validation des résultats, Benchmark

Ce benchmark est exploratoire, il vise à découvrir les avantages et inconvénients des deux méthodes.

Variables utilisées, mesurées

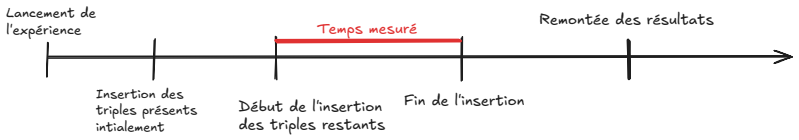
Notre benchmark vise à représenter plusieurs situations qui pourraient se présenter dans des conditions réelles, respectivement :

- Un grand ajout de données dans une base vide (équivalente à une initialisation à partir d'une source existante)
- Un grand ajout de données dans une base préremplie
- Des ajouts plus petits dans une base de données préremplie (équivalente à des modifications sur un système existant)

Ainsi, nous utiliserons trois variables différentes dans nos benchmarks :

- Le nombre de triplets pré-insérés en base de données
- Le nombre de triplets à insérer en base de données
- La méthode d'insertion utilisée : l'algorithme en one-shot, notre proposition (`naive`), notre proposition optimisée (`naive_batch`)

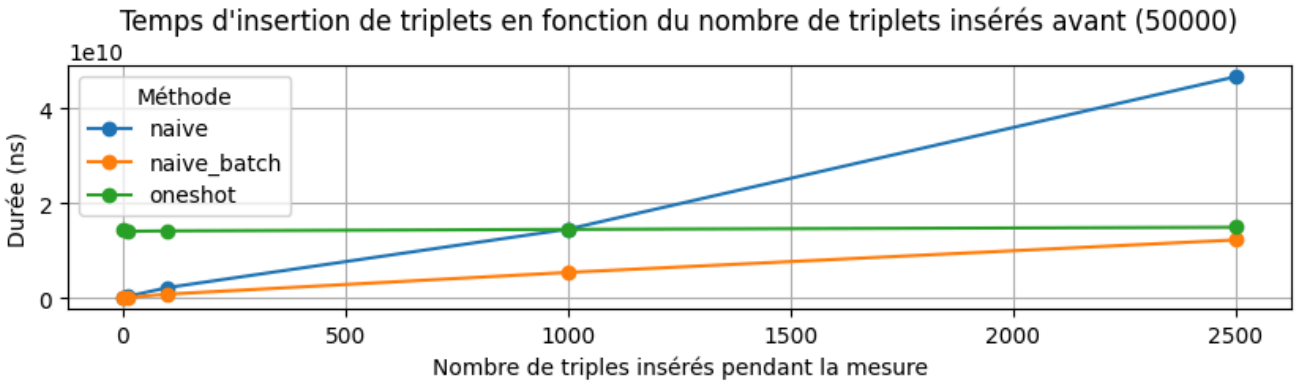
Pour chaque ensemble de valeurs, nous lançons une expérience que l'on peut représenter ainsi :



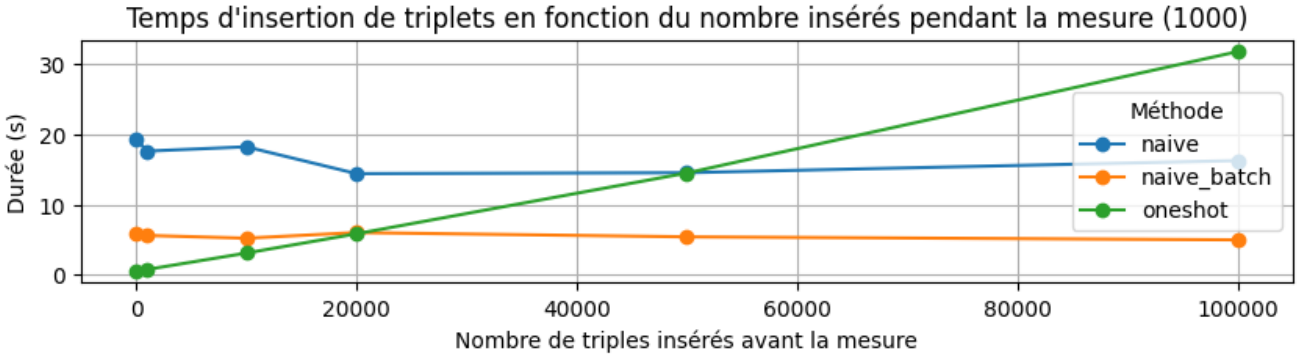
Le temps mesuré est la variable importante de cette expérience. Nous notons d'autres variables, dont l'activité sur disque afin d'isoler des situations dans lesquelles l'environnement de virtualisation serait cause de retards dans l'exécution à cause d'une trop grade latence dans les accès disques.

Résultats

Nous présenterons deux visualisations différentes afin de mieux comprendre le fonctionnement de ces algorithmes : une première visualisation montre les temps d'insertion en considérant que la base a déjà été initialisée avant avec un certain nombre de triplets, et une deuxième modélisation s'intéressera aux temps d'insertion d'un nombre fixé de triplets selon la taille initiale de la base de données.



Dans cette situation, nous pouvons voir que sur une base relativement conséquente pré-initialisée, l'algorithme en one-shot a un temps d'exécution constant, explicable par le fait que 2500 triplets est largement inférieur aux 50000 déjà présents. On note que l'algorithme incrémental semble être en complexité linéaire selon le nombre de triplets à insérer. De plus, on pourra noter qu'il existe un point d'intersection entre la courbe de l'algorithme en one-shot et celle de l'algorithme incrémental, que nous explorerons avec le graphique suivant.



Cette visualisation représente bien la question du point d'équilibre qui représente la bascule entre le moment ou l'algorithme incrémental est plus intéressant à utiliser que l'algorithme en one-shot. Dans ce graphique, nous voyons que pour insérer 1000 triplets, il est intéressant d'employer l'algorithme incrémental à partir du moment ou la base de données contient initialement plus de 20000 triplets. De plus, il est intéressant de noter que l'algorithme incrémental utilise un temps relativement constant quel que soit la taille totale de la base de données, contrairement à l'algorithme en one-shot qui est en complexité linéaire sur la taille totale de la base de données.

Conclusion

Dans ce projet de recherche, nous avons eu l'occasion de mettre en oeuvre un algorithme incrémental d'insertion de triplets dans une base de données RDF en incluant la question de la gestion des permissions telle que présentée dans la thèse de Tarek SAYAH.

Afin de répondre aux problématiques de recalculs conséquents de ces permissions, un algorithme incrémental à été proposé, dont des résultats de benchmarks exploratoires semblent montrer une meilleure performance dans le cadre d'insertions de petits nombres de triplets dans une base déjà initialisée.

Pistes d'évolution

Plusieurs sont envisageables au sortir de ce rapport : il convient tout d'abord de rappeler la possibilité d'optimiser la méthode existante. De plus, il est envisageable de prouver formellement l'algorithme afin de valider des propriétés de correction sur l'algorithme d'insertion. De plus, il pourrait être intéressant de développer l'algorithme de suppression de triplets, dont des ébauches sont proposées dans les sources du projet, sans plus de développements.

Annexes

Grammaire d'un ensemble de règles

```
grammar policy;

// Lexer grammar

LITTERAL : '<' [a-zA-Z0-9.:/\-_#]+ '>';
VAR : QUESTION_MARK [a-zA-Z0-9]+;

POLICY : 'POLI';

AUTHSCOPE : 'AUTHSCOPE';
DEFAULT : 'DEFAULT';
GRAPH : 'GRAPH';

CHOICE : 'CHOICE';
GRANT : 'GRANT';
DENY : 'DENY';
WHERE : 'WHERE';
DOT : '.';
QUESTION_MARK : '?';

POLICY_NAME : [a-zA-Z0-9]+;
CHOICE_NAME : [a-zA-Z0-9.]+;

WS: [ \t\n]+ -> skip;

// Parser Grammar

start : policy EOF;

policy : policy_intro authscope_intro choice_intro rule+;

policy_intro : POLICY POLICY_NAME;

authscope_intro : AUTHSCOPE DEFAULT GRAPH;

choice_intro : CHOICE CHOICE_NAME;
```

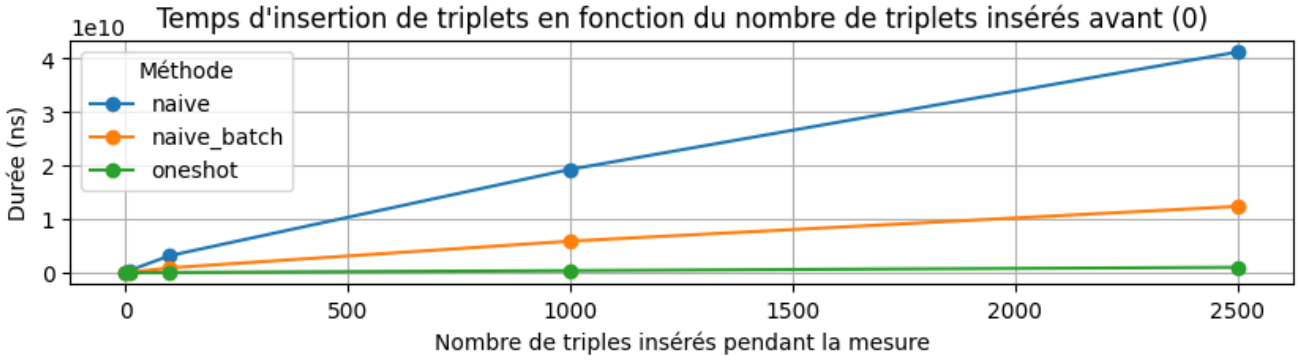
```
rule: permission term term term (where_expr | DOT);

permission : GRANT | DENY;

where_expr: WHERE (where_conj)+;

where_conj: term term term DOT;

term : VAR | LITTERAL;
```

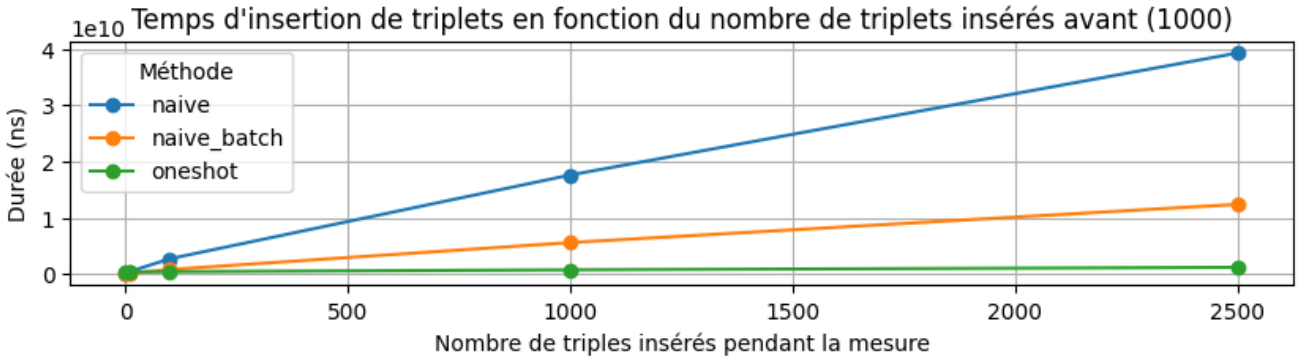
Lien du repository Gitlab

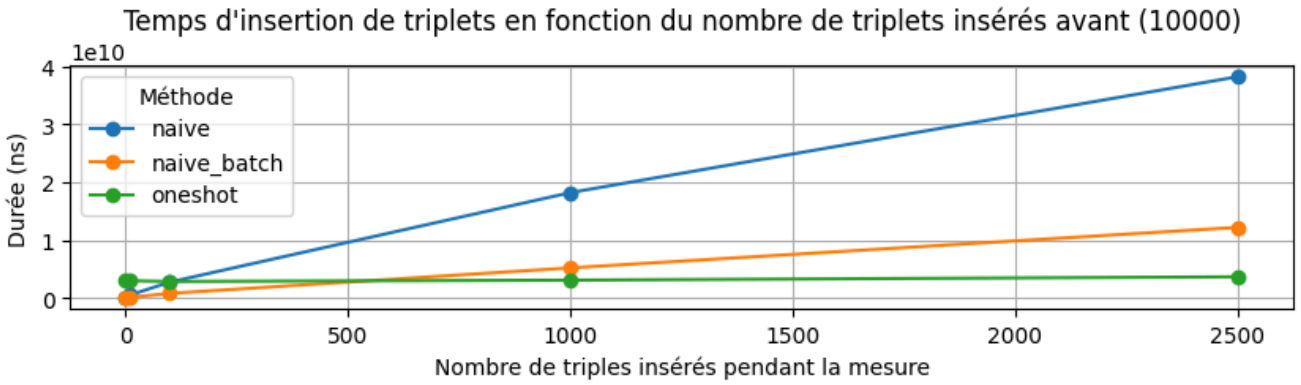
L'ensemble du code proposé pour ce projet est disponible à cette URL https://forge.univ-lyon1.fr/p2105678/mifor-rdf_perms

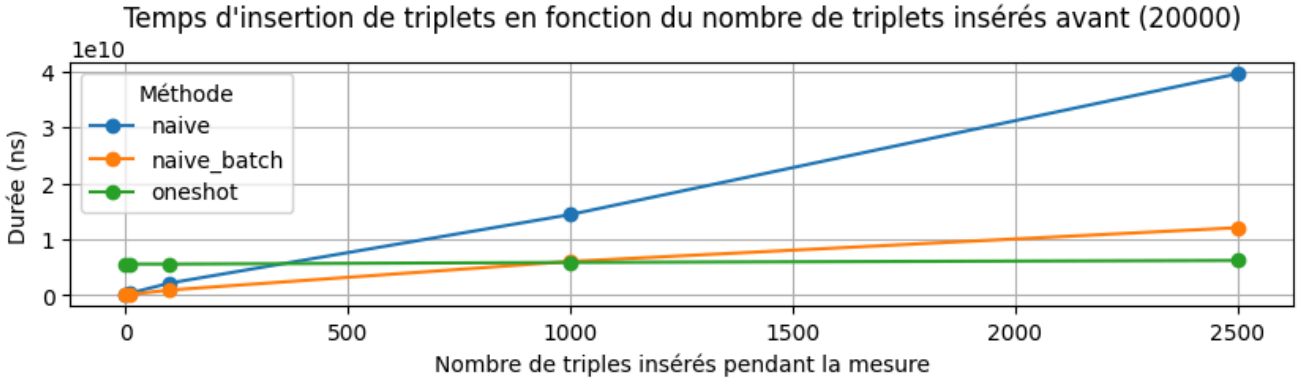
Graphiques des résultats du benchmark

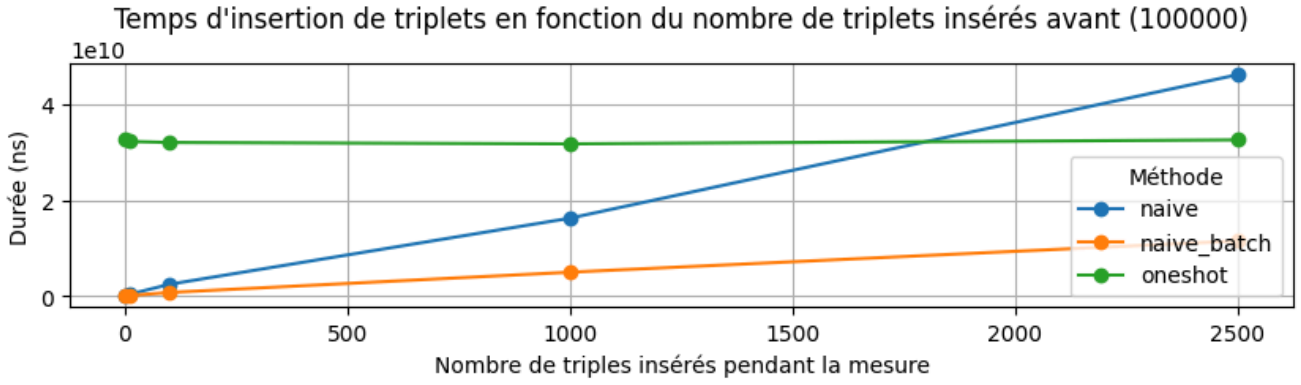
Graphiques d'évolution des temps d'insertion en fixant la taille initiale des graphes



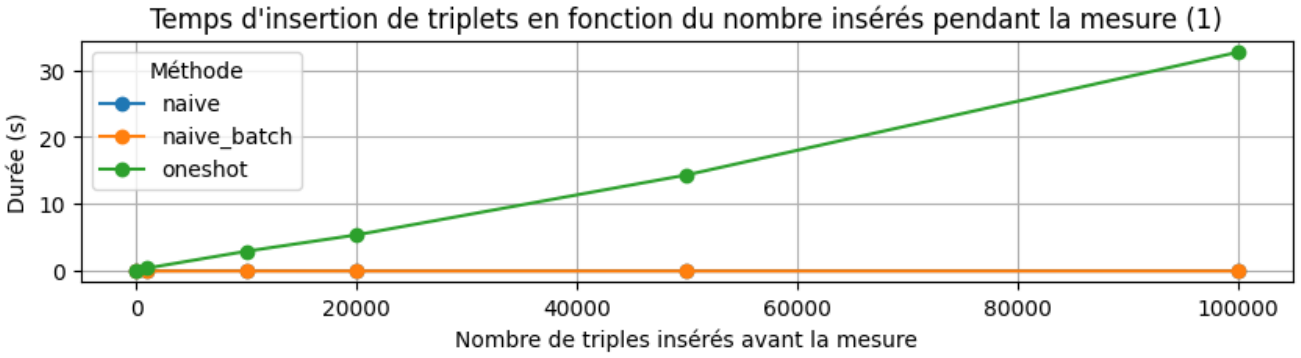


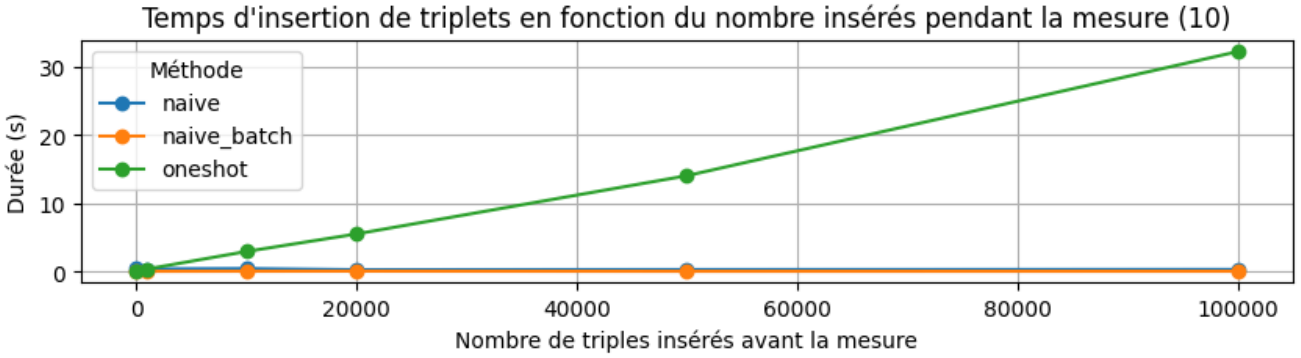


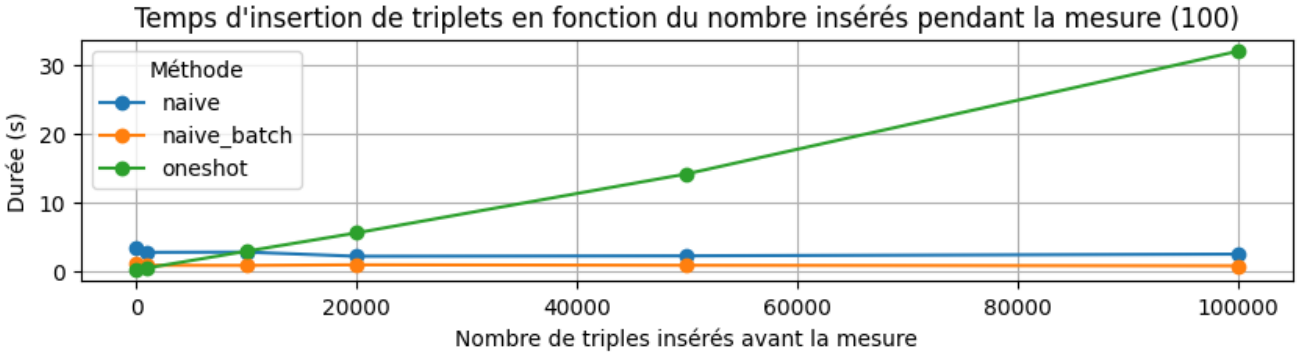


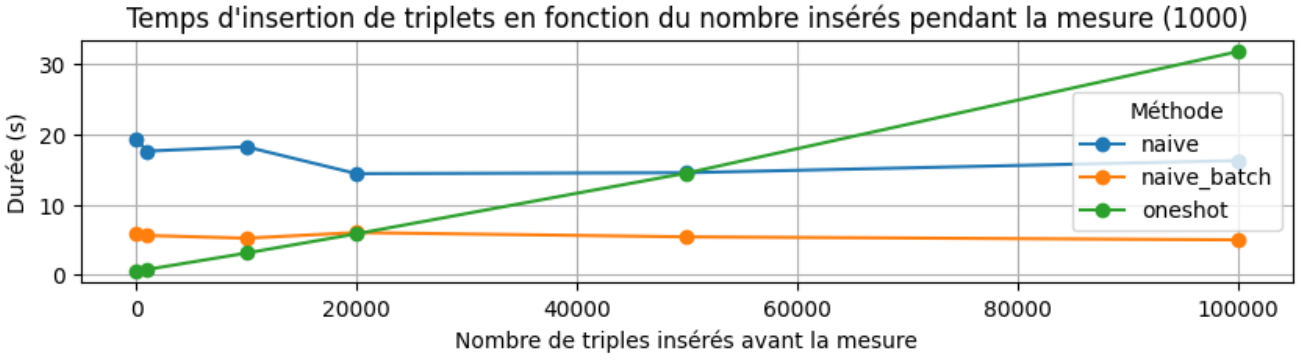


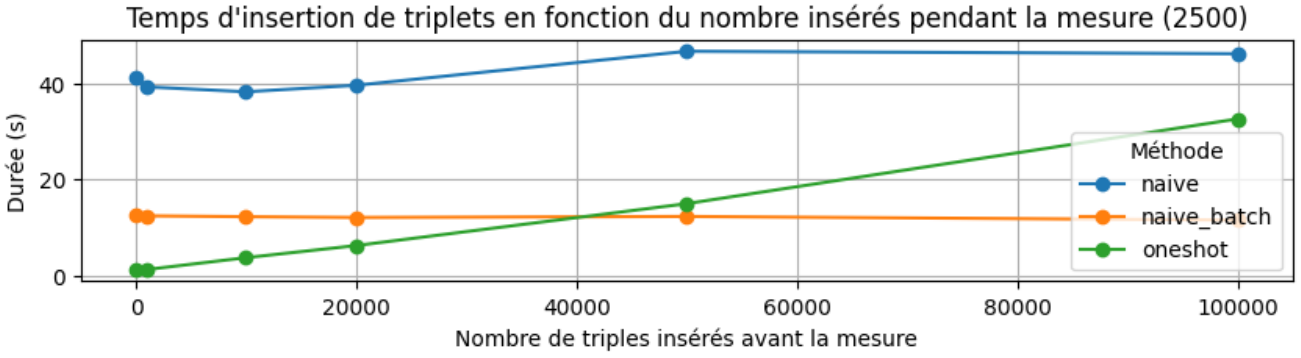
Graphiques d'évolution des temps d'insertion en fixant le nombre de triplets insérés


Temps d'insertion de triplets en fonction du nombre insérés pendant la mesure (1)
Méthode
naive
naive_batch
oneshot
Durée (s)
0
10
20
30
0
20000
40000
60000
80000
100000
Nombre de triples insérés avant la mesure


Temps d'insertion de triplets en fonction du nombre insérés pendant la mesure (10)
Méthode
naive
naive_batch
oneshot
Durée (s)
0
10
20
30
0
20000
40000
60000
80000
100000
Nombre de triples insérés avant la mesure


Temps d'insertion de triplets en fonction du nombre insérés pendant la mesure (100)
Méthode
naive
naive_batch
oneshot
Durée (s)
0
10
20
30
0
20000
40000
60000
80000
100000
Nombre de triples insérés avant la mesure


Temps d'insertion de triplets en fonction du nombre insérés pendant la mesure (1000)
Méthode
naive
naive_batch
oneshot
Durée (s)
0
10
20
30
0
20000
40000
60000
80000
100000
Nombre de triples insérés avant la mesure


Temps d'insertion de triplets en fonction du nombre insérés pendant la mesure (2500)
Méthode
naive
naive_batch
oneshot
Durée (s)
0
20
40
0
20000
40000
60000
80000
100000
Nombre de triples insérés avant la mesure